Detecting Subtle Departures from Randomness

Vincent Granville, Ph.D.
vincentg@MLTechniques.com
www.MLTechniques.com
Version 1.1, July 2022

Abstract

I discuss a new test of randomness for pseudo random number generators (PRNG), to detect subtle patterns in binary sequences. The test shows that congruential PRNGs, even the best ones, have flaws that can be exacerbated by the choice of the seed. This includes the Mersenne twister used in many programming languages including Python. I also show that the digits of some numbers such as $\sqrt{2205}$, conjectured to be perfectly random, fail this new test, despite the fact that they pass all the standard tests. I propose a methodology to avoid these flaws, implemented in Python. The test is particularly useful when high quality randomness is needed. This includes cryptographic and military-grade security applications, as well as synthetic data generation and simulation-intensive Markov chain Monte Carlo methods. The origin of this test is in number theory and connected to the Riemann Hypothesis. In particular, it is based on Rademacher stochastic processes. These random multiplicative functions are a number-theoretic version of Bernoulli trials. My article features state-of-the-art research on this topic, as well as an original, simple, integer-based formula to compute square roots to generate random digits. It is offered with a Python implementation that handles integers with millions of digits.

Contents

1 Introduction

Let $\chi(\cdot)$ be a function defined for strictly positive integers, with $\chi(1) = 1$ and $\chi(ab) = \chi(a)\chi(b)$ for any integers $a, b > 0$. Such a function is said to be completely multiplicative [Wiki]. Here we are interested in the case where χ takes on two possible values: $+1$ and -1. The core of my methodology is based on the following, well-known identity:

$$\sum_{k=1}^{\infty} \chi(k)k^{-z} = \prod_{p\in P} \frac{1}{1-\chi(p)p^{-z}}. \tag{1}$$

The product is over all prime integers ordered by increasing values: $P = \{2, 3, 5, 7, 11, \dots\}$ is the set of all primes. Such a product is called an Euler product [Wiki]. The series on the left-hand side is called a Dirichlet series [Wiki]. The argument $z = \sigma + it$ is a complex number. You don't need to know anything about complex numbers to understand this article. The only important fact is that the series or product converges only if σ – the real part of z – is large enough, typically larger than 0, $\frac{1}{2}$ or 1, depending on χ. If χ is a constant function, thus equal to 1, then the product and series converge to the Riemann zeta function $\zeta(z)$ [Wiki] if $\sigma > 1$.

For primes p, let the $\chi(p)$'s be independent random variables, with $\mathrm{P}[\chi(p) = 1] = \mathrm{P}[\chi(p) = -1] = \frac{1}{2}$. The product, denoted as $L_P(z, \chi)$, is known as a Rademacher random multiplicative function, see [4], [5] and [8]. If

z is a complex number, we are dealing with complex random variables [Wiki]. From the product formula and the independence assumption, it is easy to obtain

$$\mathrm{E}[L_P(z,\chi)] = \prod_{p\in P} \mathrm{E}\left[\frac{1}{1-\chi(p)p^{-z}}\right] = \prod_{p\in P} \frac{1}{1-p^{-2z}} = \zeta(2z). \tag{2}$$

Thus the expectation is finite if $\sigma = \Re(z) > \frac{1}{2}$. A similar argument can be used for the variances.

Now let us replace the random variables $\chi(p)$ by pseudo-random numbers, taking the values $+1$ or -1 with probability $\frac{1}{2}$. If these generated numbers are "random enough", free of dependencies, then one would expect them to satisfy the laws of Rademacher random multiplicative functions. The remaining of this article explores this idea in details, with a focus on applications.

2 Pseudo-random numbers

There is no formal definition of pseudo-random numbers. Intuitively, a good set of pseudo-random numbers is a deterministic binary sequence of digits that satisfies all statistical tests of randomness. Of course, it makes no sense to talk about randomness if the sequence contains very few digits, say one or two. So pseudo-random numbers (PRN) are associated with infinite sequences, even though in practice one only uses finite sub-sequences.

A rigorous definition of PRN sequences requires the convergence of the empirical joint distribution [Wiki] of any finite sub-sequence of m digits, to the known theoretical value under the assumption of full randomness. Let the PRN sequence be denoted as $\{d(k)\}$ with $k = 1, 2$ and so on. A sub-sequence of m digits is defined by its indices, denoted as $i_1, i_2, \ldots i_m$. The convergence of the empirical distribution means that regardless of the indices $0 \le i_1 < i_2 < \ldots$ we have:

$$\lim_{n\to\infty} \frac{1}{n} \sum_{k=1}^{n} I\Big[d(k+i_1) = k_1, d(k+i_2) = k_2, \ldots, d(k+i_m) = k_m\Big] = 2^{-m} \tag{3}$$

for any $(k_1, k_2, \ldots, k_m)$ in $\{-1, +1\}^m$. Here I is the indicator function: $I[A] = 1$ if A is true, otherwise $I[A] = 0$. The following number λ is of particular interest:

$$\lambda = \sum_{k=1}^{\infty} d'(k) \cdot 2^{-k}, \quad \text{with } d'(k) = \frac{1+d(k)}{2} \in \{0, 1\}. \tag{4}$$

Thus the $d'(k)$'s are the binary digits of the number λ, with $0 \le \lambda \le 1$.

The connection between the multiplicative function $\chi(\cdot)$ in Formula (1) and the $d(k)$'s is as follows. Let denote the k-th prime as p_k, with $p_1 = 2$. Then $d(k) = \chi(p_k)$. The traditional definition of PRN's is equivalent to requiring λ to be a normal number in base 2 [Wiki]. I introduce a stronger criterion of randomness in section 2.1

2.1 Strong pseudo-random numbers

Convergence of the empirical joint distributions, as defined by Formula (3), has a few important implications. The Kolmogorov-Smirnov test [Wiki], the Berry-Esseen inequality [Wiki] and the law of the iterated algorithm [Wiki] can be applied to the PRN sequence $\{d(k)\}$. These three fundamental results provide strong limitations on the behavior of finite PRN sequences. If a sequence $\{d(k)\}$ or its representation by the number λ does no stay within these limits, then it does not emulate pure randomness. However, some quasi-random PRN sequences, with weak dependencies, meet these requirements yet are not truly "random". For instance, a number can be normal in base 2 yet have digits that exhibit some dependencies, see here. The purpose of this section is to introduce stronger requirements in order to catch some of these exceptions. This is where the multiplicative function $\chi(\cdot)$ comes into play.

The function $\chi(\cdot)$, initially defined for primes p, is extended to all strictly positives integers via $\chi(ab) = \chi(a)\chi(b)$. Because the $\chi(p)$'s are independent among prime numbers (by construction), the full sequence $\{\chi(k)\}$ over all k must behave in a certain way. Obviously, if k is a square integer, $\chi(k) = 1$. But if k is not a square, we still have $P[\chi(k) = 1] = P[\chi(k) = -1] = \frac{1}{2}$. For instance, $\chi(4200) = \chi(4)\chi(25)\chi(6)\chi(7) = \chi(6)\chi(7)$. Since the product of two independent random variables with Rademacher distribution [Wiki] has a Rademacher distribution, it follows that $\chi(6) = \chi(2)\chi(3)$ has a Rademacher distribution, and thus $\chi(4200) = \chi(6)\chi(7)$ also has a Rademacher distribution. So, the $\chi(k)$'s are identically distributed with zero mean, unless k is a square integer. However, they are not independently distributed, even after removing square integers, or even if you only keep square-free integers [Wiki].

I now define three fundamental functions, which are central to my new test of randomness. First, define the following sets:

- $S_1(n)$ contains all prime integers $\leq n$.
- $S_2(n)$ contains all positive square-free integers $\leq n$.
- $S_3(n)$ contains all positive non-square integers $\leq n$.

So each of these sets contains at most n elements. Then, define the three functions as

$$L_1(n) = \sum_{k \in S_1(n)} \chi(k), \quad L_2(n) = \sum_{k \in S_2(n)} \chi(k), \quad L_3(n) = \sum_{k \in S_3(n)} \chi(k). \tag{5}$$

Now, I can introduce my new test of randomness.

2.1.1 New test of randomness for PRNGs

Let $d(1), d(2), \ldots$ be a sequence of integer numbers, with $d(k) \in \{-1, 1\}$ and $P = \{p_1, p_2, \ldots\}$ be the set of prime numbers. The goal is to test how random the sequence $\{d(k)\}$ is, based on the first n elements $d(1), \ldots, d(n)$. The algorithm is as follows.

- Step 1: Set $\chi(p_k) = d(k)$, where p_k is the k-th prime number ($p_1 = 2$).
- Step 2: For $k \notin P$ with prime factorization $k = p_1^{a_1} p_2^{a_2} \cdots$, set $\chi(k) = \chi^{a_1}(p_1)\chi^{a_2}(p_2)\cdots$, with $\chi(1) = 1$.
- Step 3: Using Formula (5), compute $L_3^*(n) = |L_3(n)|/\sqrt{n \log\log n}$.
- Step 4: If $L_3^*(n) < 0.5$ or $L_3^*(n) > 1.5$, the sequence $\{d(k)\}$ (the first n elements) lacks true randomness.

This test is referred to as the "prime test". Let's illustrate step 2 with $k = 4200$: since $4200 = 2^3 \cdot 3 \cdot 5^2 \cdot 7$, we have $\chi(4200) = \chi^3(2)\chi(3)\chi^2(5)\chi(7) = \chi(2)\chi(3)\chi(7)$.

Some non-random sequences may pass the prime test. So you should never use this test alone to decide whether a sequence is good enough. Also, the standardization of $L_3(n)$, using the $\sqrt{n \log\log n}$ denominator, is not perfect, but good enough for all practical purposes, assuming $10^4 < n < 10^{15}$. This test can detect departure from randomness that no other test is able to uncover. I discuss practical examples and a Python implementation later in this article.

A PRN sequence that satisfies (3) and passes all the existing tests, including the prime test, is called strongly pseudo-random. The corresponding real number λ defined by Formula (4) is called strongly normal. It should not be difficult to prove that almost all numbers are strongly normal. Thus almost all PRN sequences are strongly pseudo-random. Yet creating one that can be proved to be strongly pseudo-random is as difficult as proving that a given number is normal (and a fortiori, strongly normal). Interestingly, none of the sequences produced by congruential random number generators [Wiki] are strongly pseudo-random, for the same reason that no rational number is normal: in both cases, $d(k)$ is periodic.

Modern test batteries include the Diehard tests [Wiki] published in 1995, and the TestU01 framework [Wiki], introduced in 2007.

2.1.2 Theoretical background: the law of the iterated logarithm

The prime test checks whether the multiplicative function $\chi(k)$ derived from $\{d(k)\}$, satisfies a particular version of the law of the iterated logarithm [Wiki]. Truly random sequences $\{d(k)\}$ satisfy that law. Since $\{\chi(k)\}$ is multiplicative and thus non-random, the law of the iterative algorithm must be adapted to take care of the resulting dependencies. In particular, the $\sqrt{n \log\log n}$ weight used to standardize $L_3(n)$ provides only an approximation, good enough for all practical purposes. The exact weight is discussed in [5] and [8].

Many sequences $\{d(k)\}$ satisfy the basic law of the iterated algorithm without the prime number / Euler product apparatus introduced in this article. This is the case for most of the sequences studied here. However, by looking at $\chi(k)$ rather than the original $d(k)$, we are able to magnify flaws that are otherwise undetectable by standard means. An example is the Mersenne twister implemented in Python, passing the standard test, but failing the prime test when the seed is set to 200.

2.1.3 Connection to the Generalized Riemann Hypothesis

The Generalized Riemann Hypothesis (GRH) [Wiki] is one of the most famous unsolved problems in mathematics. It states that the function $L(z, \chi)$ defined by Formula (1) has no root if $\frac{1}{2} < \sigma = \Re(z) < 1$. Here $z = \sigma + it$ is a complex number, and $\sigma = \Re(z)$ is its real part. It applies to a particular class of functions $\chi(\cdot)$, those that are "well behaved". Of course, without any restriction on $\chi(\cdot)$, there are completely multiplicative functions [Wiki] known to satisfy GRH. For instance, the function defined by $\chi(p_{2k}) = -1, \chi(p_{2k+1}) = 1$ where p_k is the k-th prime number. The corresponding $L(z, \chi)$ has no root if $\sigma > \frac{1}{2}$ because the product converges for $\sigma > 0$, and of course, the product has no root. The sequence $\{\chi(p_k)\}$ is obviously non-random as it perfectly alternates, and thus I labeled it `CounterExample` in Table 1.

If the Euler product in Formula (1) converges for some $\sigma > \sigma_0$, it is equal to its series expansion when $\sigma > \sigma_0$, it converges for all $\sigma > \sigma_0$, and $L(z, \chi)$ satisfies GRH when $\sigma > \sigma_0$. When $\sigma < 1$, the convergence is conditional [Wiki], making things more difficult. Another example that trivially satisfies GRH if $\sigma > \frac{1}{2}$ is when $\chi(\cdot)$ is a random variable with $P[\chi(p) = 1] = P[\chi(p) = -1] = \frac{1}{2}$ for primes p. In this case convergence means that $L(z, \chi)$ has finite expectation as proved by Formula (2), and finite variances. This function $\chi(\cdot)$ is called a random Rademacher multiplicative function, see [6]. Here the $\chi(p)$'s are identically and independently distributed over the set of all primes, and the definition is extended to all strictly positive integers with the formula $\chi(ab) = \chi(a)\chi(b)$.

So, if we were able to find a "nice enough" pseudo-random yet deterministic function $\chi(\cdot)$ with convergence of the Euler product when $\sigma > \sigma_0$ for some $\sigma_0 < 1$, a function $\chi(\cdot)$ that is random enough over the primes (like its sister, the truly stochastic version) to guarantee the convergence of the product, then it would be a major milestone towards proving GRH. Convergence of the product would imply that:

- $L(z, \chi)$ is analytic [Wiki], because the product is equal to its series expansion, which trivially satisfies the Cauchy-Riemann equations [Wiki],
- $L(z, \chi)$ has no root if $\sigma > \sigma_0$ since the product has no root.

As discussed, examples that are "not so nice" exist; "nice enough" means that $L(z, \chi)$ satisfies a Dirichlet functional equation [Wiki]. Typically, "nice enough" means that $L(z, \chi)$ is a Dirichlet-L function [Wiki]. Besides the Riemann zeta function where $\chi(p) = 1$ is the trivial Dirichlet character, the most fundamental example is when $\chi = \chi_4$ is the non-trivial Dirichlet character modulo 4 [Wiki]. This example is featured in Figure 1 where $\sigma = \frac{1}{2}$ (left plot) and contrasted with a not so nice example on the right plot, corresponding to a pseudo-random sequence $\{\chi(p_k)\}$. The χ_4 example is referred to as `Dirichlet4` in Table 1. Note that for $\sigma = \frac{1}{2}$, $L(z, \chi_4)$ has infinitely many roots just like the Riemann zeta function, though its roots are different: this is evident when looking at the left plot in Figure 1. The (conjectured) absence of root is when $\frac{1}{2} < \sigma < 1$.

I went as far as to compute the Euler product for $L(z, \chi_4)$ when $z = \sigma = 0.99$. It nicely converges to the correct value, without the typical growing oscillations associated to lack of convergence, agreeing with the value computed using the series expansion in Formula (1). It means that the product converges at least for all z with $\sigma = \Re(z) \geq 0.99$. Thus there is no root for $L(z, \chi_4)$ if $\sigma \geq 0.99$. This would be an immense milestone compared to the best known result (no root if $\sigma \geq 1$) if it was theoretically possible to prove the convergence in question, supported only by empirical evidence. Convergence implies that the sequence $\{\chi_4(p_k)\}$ is random enough over the primes p_1, p_2 and so on. That is, the gap between $+1$ and -1 in the sequence never grows too large (that is, runs of same value can only grow so fast), and the proportion of $+1$ and -1 tends to $\frac{1}{2}$ fast enough, despite the known Chebyshev's bias [Wiki]. The fact that the proportion eventually converges to $\frac{1}{2}$ when using more and more terms in the sequence, is a consequence of Dirichlet's theorem [Wiki]. This is how close we are – or you may say how far – to proving GRH.

There are other "nice functions" $\chi(\cdot)$ that fit within the GRH framework. For instance, with primes that are not integers, such as Beurling primes [7] (discussed later in this article) or Gaussian primes [Wiki]. For a general family of such functions, see the Dedekind zeta function [Wiki]. For a general introduction to the Riemann zeta function and related topics, see [1] and [9].

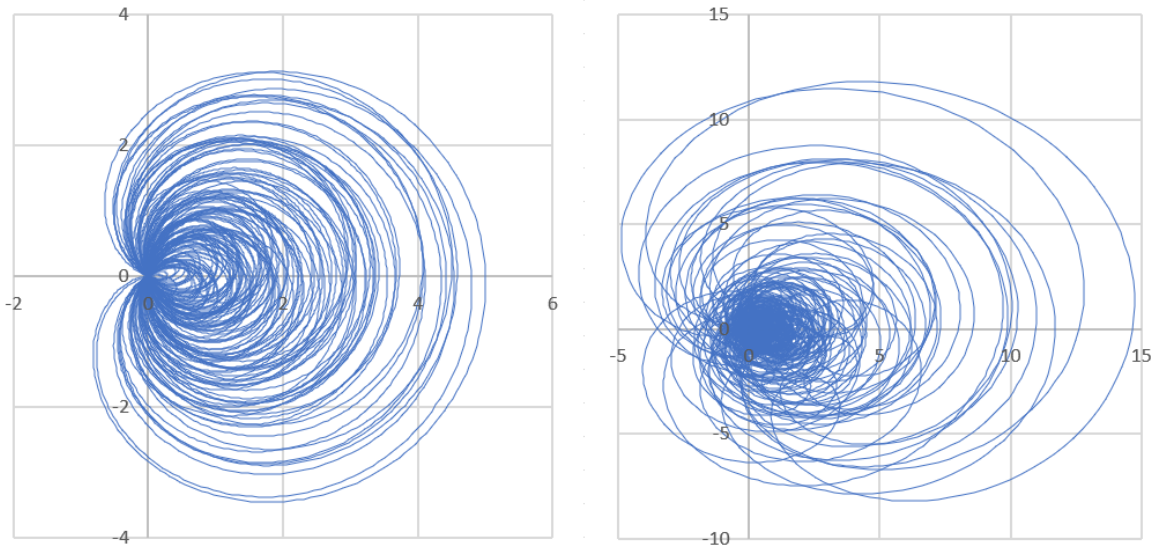


Figure 1: Orbit of $L(z, \chi)$ at $\sigma = \frac{1}{2}$, with $0 < t < 200$ and $\chi = \chi_4$ (left) versus pseudo-random χ (right)

2.2 Testing well-known sequences

The binary sequences analyzed here are denoted as $\{d(k)\}$, with $d(k) \in \{-1, +1\}$ and $k = 1, 2$ and so on. The tests, restricted to $d(k) \leq n$, are based on $L_3^*(n) = L_3(n)/\sqrt{n \log\log n}$, with $L_3(n)$ defined by (5). Again, $\chi(p_k) = d(k)$ for prime numbers, and $\chi(\cdot)$ is extended to non-primes positive integers via $\chi(ab) = \chi(a)\chi(b)$. Finally, p_k is the k-th prime with $p_1 = 2$. In my examples, $n = 20{,}000$ in Table 1, and $n = 80{,}000$ in Figures 2 and 3.

The fact that a sequence fails the test for a specific n does not mean it fails for all n. The success or failure also depends on the seed (the initial conditions). Some seeds require many iterations – that is, a rather large n – before randomness kicks in. The test should not be used for small n, say $n < 1000$. Finally, passing the test does not mean that the sequence is random enough. I provide examples of poor PRNGs that pass the test. Typically, to assess the randomness character of a sequence, one uses a battery of tests, not just one test. However, the prime test can detect patterns that no other one can. In some sense, it is a last resort test.

The sequences investigated here fall into four types:

- Discrete chaotic dynamical systems [Wiki]. In one dimension, many of these systems are driven by a recursion $x_{k+1} = g(x_k)$, where g is a continuous mapping from $[0, 1]$ onto $[0, 1]$. The initial value x_0 is called the seed. Typically, $d(k) = 1$ if $x_k < 0.5$, otherwise $d(k) = -1$. The logistic map [Wiki], with $g(x) = 4x(1 - x)$, is labeled `Logistic` in Table 1 as well as in the Pyhton code in section 3.2. The shift map in base b, defined by $g(x) = bx - \lfloor bx \rfloor$ where the brackets represent the integer part function, here with $b = 3$, is labeled `Base3`. The case $b = 2$ is known as the dyadic map [Wiki]. The number $\lfloor bx_k \rfloor$ is the k-digit of the seed x_0, in base $b > 1$. In particular, if x_0 is a rational number, then the sequence $\{d(k)\}$ is periodic, and thus non random. Even in the best-case scenario (using a random seed), the sequence $\{d(k)\}$ is auto-correlated. These dynamical systems are studied in detail in my book on probabilistic properties of numeration systems [3].
- Mersenne twister [Wiki] as implemented in the Python function `random.random()`. This congruential PRNG is also another type of dynamical system, though technically "non-chaotic" because the sequence $\{x_k\}$ is periodic. It emulates randomness quite well as the period is very large. Likewise, the shift map with a large base b and a bad seed x_0 (a rational number resulting in periodicity) will emulate randomness quite well if $x_0 = q/p$, where p, q are integers and p is a very large prime. Both in the table and in the figures, I use the label `Python` for the Mersenne twister. It fails the prime test for various seeds, especially if the seed is set to 200. See also section 3.1.
- Number theoretic sequences related to the distribution of prime numbers or Beurling primes. Sequences of this type are labeled `Dirichlet4`. The main one, with the seed set to 3, corresponds to $\chi(p_k) = +1$ if $p_k \equiv 1 \bmod 4$ and $\chi(p_k) = -1$ if $p_k \equiv 3 \bmod 4$. Here $\chi(2) = 0$. This function, denoted as χ_4, is the non-trivial Dirichlet character modulo 4 [Wiki]. The sequence barely fails the L_3^* test; the full function χ_4 defined over all positive integers (not just the non-square integers) is periodic. The sister sequence (with the seed set to 1) has $\chi(p) = -\chi_4(p)$ if p is prime. Sequences based on Beurling primes (a generalization of prime numbers to non-integers) are not included yet, but discussed in my upcoming article on the Riemann Hypothesis, to be published here. The Python code in section 3.2 can easily be adapted to handle them. The `DirichletL.py` code posted here on my GitHub repository, includes Beurling primes. These numbers are studied in Diamond [2] and Hilberdink [7].
- Binary digits of quadratic irrational numbers. I use a simple, original recursion to compute these digits: see code description in section 3.3. The Python code painlessly handles the very large integers involved in these computations. Surprisingly, $\sqrt{2}$ passes the prime test as expected, but $\sqrt{2205}$ does not. Sequences based on these digits are labeled `SQRT` in this document.

The dyadic map is impossible to implement in Python due to the way computations are performed in the CPU: the iterates $\{x_k\}$ are (erroneously) all equal to zero after about 45 iterations. This is why I chose the shift map with $b = 3$. In this case, the iterates are also all wrong after 45 iterations due to propagating errors caused by limited machine precision (limited to 32 bits). Even with 64 or any finite number of bits, the problem persists. However, with $b = 3$, the x_k's keep oscillating properly and maintain their statistical properties forever (including randomness or lack of), due to the ergodicity [Wiki] of the system. The result is identical to using a new seed every 45 iterations or so.

The dyadic map with $b = 2$, in principle, could be used to compute the binary digits of the seed $x_0 = \sqrt{2}$, but because of the problem discussed, it does not work. Instead, I use a special recursion to compute these digits. If you replace $b = 2$ by $b = 2 - 2^{-31}$ (the closest you can get to avoid complete failure) the x_k's produced by the Python code, even though also completely wrong after 45 iterations, behave as expected from a statistical point of view: this is a workaround to using $b = 2$. The same problem is present in other programming languages.

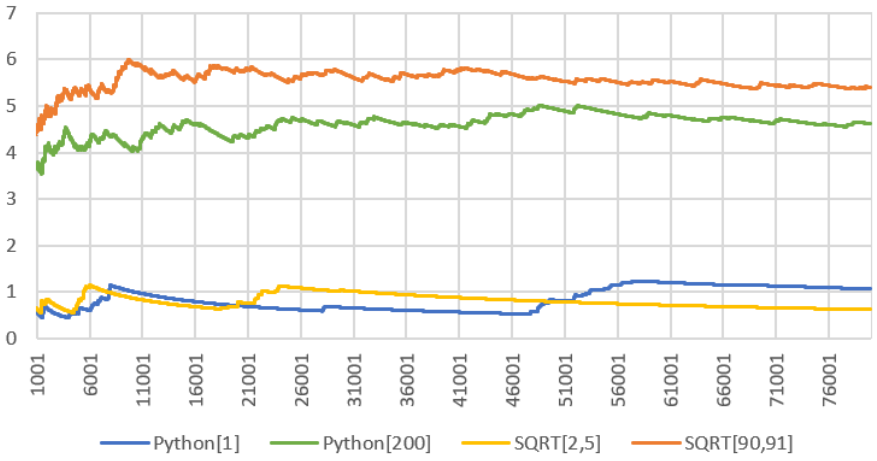


Figure 2: $L_3^*(n)$ test statistic for four sequences: Python[200] and SQRT[90,91] fail

2.2.1 Reverse-engineering a pseudo-random sequence

Many of the sequences defined by a recursion $x_{k+1} = g(x_k)$, where x_0 is the seed, can be reverse-engineered, and are thus unsafe to use when security is critical. This includes sequences produced by congruential PRNGs. By reverse engineering, I mean that if you observe m consecutive digits, you can easily compute all the digits, and thus correctly "guess" the whole sequence. In the case of the Mersenne twister, $m = 624$ is conjectured to be the smallest possible value even though the period is $2^{19,937} - 1$, see here. For the shift map in base b, while x_k is asymptotically uniformly distributed on $[0, 1]$ if x_0 is random, the vectors (x_k, x_{k+1}) lie in a very specific configuration: $x_{k+1} - x_k$ is a small integer, making the sequence $\{x_k\}$ anything but random. As a result, for any positive integer q, the empirical autocorellation [Wiki] between $(x_1, x_2, x_3, \dots)$ and $(x_{q+1}, x_{q+2}, x_{q+3}, \dots)$ computed on the infinite sequence, is equal to $1/b^q$ if b is an integer ≥ 2. A good sequence should have zero autocorrelations for all q.

It is possible to significantly improve the base sequence $\{x_k\}$, to make it impossible to reverse-engineer. In the case of the shift map, using $d(k) = \lfloor bx_k \rfloor$ instead of x_k, results in zero autocorrelations and perfect randomness if the seed x_0 is random. A seed such as $x_0 = \sqrt{2}/2$ or $x_0 = \log 2$ is conjectured to achieve this goal. The explanation is as follows: $d(k)$ is the k-th digit of x_0 in base b. Even if you were to observe $m = 10^{50,000}$ consecutive digits of $\sqrt{2}/2$, there is no way to predict what the next digit will be, if you don't know that $x_0 = \sqrt{2}/2$. Actually even if you have that information, it is still impossible to predict the next digit. Any sequence of m digits is conjectured to occur infinitely many times at arbitrary locations for a seed such as $\sqrt{2}/2$. So given any such string of m digits (no matter how large m is), it is impossible to tell where it takes place in the infinite sequence of digits, and thus impossible to correctly predict all the subsequent digits.

However, because of machine precision (the problem discussed in section 2.2), the x_k's generated by a computer for the shift map (or any map for that matter), eventually become periodic. Thus $\{d(k)\}$ becomes periodic too. A workaround is the use exact arithmetic to compute $d(k)$, as in my Python code in section 3.3. Another solution is to use Bailey–Borwein–Plouffe formulas [Wiki] to compute the digits. There are many BBP formulas for various good transcendental seeds [Wiki] such as $x_0 = \frac{\pi}{4}$, but as far as I know, none for the subset of algebraic numbers [Wiki] such as $x_0 = \sqrt{2}/2$.

2.2.2 Illustrations

Figures 2 and 3 show the core statistics of the prime test, defined by Formula (5): $L_3^*(n)$ and $|L_3(n)|$, for n between 1000 and 80,000. If $L_3^*(n) < 0.5$ or $L_3^*(n) > 1.5$, the sequence $\{d(k)\}$ (the first n elements) lacks true randomness; it is not strongly pseudo-random. Table 1 summarizes these findings for a larger collection of sequences, computed at $n = 20{,}000$. The notation `Python[200]` corresponds to the Python implementation of the Mersenne twister, using the `random.random()` function and the seed 200, that is, `random.seed(200)`. Similarly, `SQRT[90,91]` is for the binary digits of $\sqrt{2205}$, obtained using the bivariate seed $y = 90, z = 91$ in the code in section 3.3. Not surprisingly, the sequence `Base3[0.72]` fails, as $0.72 = 18/25$ is a rational number with a small denominator. Thus $d(k) = \chi(p_k)$ is periodic with a rather small period. The column labeled `Status` in Table 1 indicates if the sequence in question fails or passes the prime test.

For convenience, I also included a type of sequences called `CounterExample`. For this type of sequences, $\chi(p_k)$ perfectly alternates between -1 and $+1$. One of the two resulting sequences $\{d(k)\}$ barely passes the test, the other one fails. Now, the `Dirichlet4` sequence with seed set to 3, has perfectly alternating $d(k)$'s

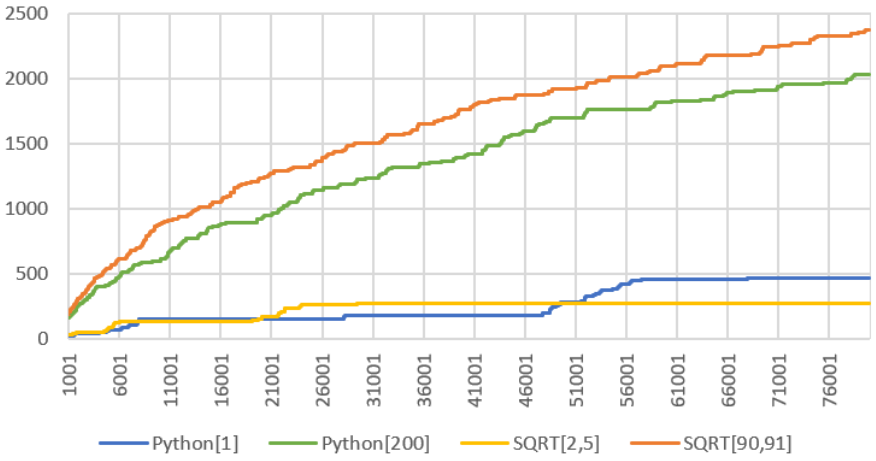


Figure 3: $|L_3(n)|$ test statistic for four sequences: Python[200] and SQRT[90,91] fail

and is thus non-random. It fails the prime test, but barely. This means that passing this test is not a guarantee of randomness. Only failing the test is a guarantee of non-randomness.

The prime test can be extended using the option `All` in the Python code. To do so, define the L_4 statistics as follows:

$$L_4(n) = \sum_{k=1}^{n} \chi(k), \quad L_4^*(n) = \frac{|L_4(n)|}{\sqrt{n \log\log n}}. \tag{6}$$

Now with L_4^* rather than L_3^*, the `Dirichlet4` sequence with the seed set to 3 would dramatically fail the prime test, rather than just barely failing. It would reveal that despite the appearances, there is something definitely non random about this sequence. Indeed, it satisfies $\chi(4k+1) = 1, \chi(4k+3) = -1$ and $\chi(2k) = 0$. The details of the L_4^* version of the prime test still need to be worked out, thus I did not include it in this article.

Finally, if you swap $-1/+1$ in the $\{d(k)\}$ sequence, the new sequence may pass the test even if the original fails (or the other way around). This is the case for the sequence `SQRT[90,91]`. Also, the L_3^* scale should be interpreted as an earthquake scale: an increase from 0.35 to 0.45, or from 1.3 to 1.8, represents a massive difference. A sequence with a low L_3^* alternates too frequently compared to a random sequence, resulting in a ratio $+1$ versus -1 too close to 50% among the $d(k)$'s. The ratio in question corresponds to the column labeled $P[d(k) = 1]$ in Table 1.

Exercise 1 – Pseudo-random sequence generated by rational numbers. Let $q_k = 2^{-\{k \log_2 3\}}$ be a rational number ($k = 1, 2$ and so on), where the brackets represent the fractional part function [Wiki]. For instance, $q_6 = 512/729$. Let M_n be the median of $\{q_1, \ldots, q_n\}$. Thus if n is odd, then M_n is the middle term after rearranging the q_k's in increasing order. Prove that (1) $M_n \to \sqrt{2}/2$ as $n \to \infty$, (2) the binary digit expansion of q_k has period $2 \cdot 3^{k-1}$ and (3) the proportion of 0 and 1 among these digits, is exactly 50/50.

Solution

Solution to (2) and (3) is found here; (3) follows from the equidistribution modulo 1 [Wiki] of the sequence $\{k \log_2 3\}$. This implies that the q_k's are distributed like 2^{-U} where U is uniformly distributed on $[0, 1]$.

3 Python code

The code in section 3.3 focuses on big integers and computing the binary digits for a class of quadratic irrational numbers, using an original, not well-known recursion, possibly published here for the first time. This code is very short, and the description is accompanied by pretty cool math. I recommend that you start looking at it, before digging into the main program in section 3.2.

The main program deals with the prime test. Before that, section 3.1 discusses some generalities related to Python and other languages, pertaining to PRNG issues and fixes.

3.1 Fixes to the faulty random function in Python

The default Python function to generate pseudo-random numbers is `random.random()`, available in the `random` library. It is based on the Mersenne twister congruential generator [Wiki], and documented here. As

Sequence	Seed	$\lvert L_3(n)\rvert$	$P[d(k)=1]$	$L_3^*(n)$	Status
Base3	0.181517	239	49.49%	1.1202	Pass
Base3	0.72	81	49.93%	0.3796	Fail
CounterExample	1	137	49.69%	0.6421	Pass
CounterExample	0	91	49.80%	0.4265	Fail
Dirichlet4	1	113	50.11%	0.7611	Pass
Dirichlet4	3	70	49.65%	0.4715	Fail
Logistic	0.181517	115	49.82%	0.539	Pass
Logistic	0.72	254	49.37%	1.1905	Pass
Python	0	220	49.71%	1.0311	Pass
Python	1	150	50.03%	0.7031	Pass
Python	2	279	49.46%	1.3077	Pass
Python	4	365	50.81%	1.7108	Fail
Python	100	386	49.10%	1.8092	Fail
Python	200	922	52.29%	4.3214	Fail
Python	500	258	49.67%	1.2093	Pass
SQRT	$(2, 5)$	146	49.63%	0.6843	Pass
SQRT	$(90, 91)$	1236	53.07%	5.7932	Fail

Table 1: $L_3^*(n)$, for various sequences ($n = 20{,}000$); "Fail" means failing the prime test

discussed in section 2.2, it is not suitable for cryptographic and other applications where pure randomness is critical. Indeed, the documentation comes with the following warning: "The pseudo-random generators of this module should not be used for security purposes".

One way to improve `random.random()` is to avoid particularly bad seeds, such as 200 or 4, in the `random.seed()` call. You may also use binary digits of some quadratic irrational numbers [Wiki], using the Python code in section 3.2. Again, it is a good idea to check, using the prime test proposed in this article, which irrational numbers to avoid. Also this method may be slow, as it involves working with very big integers. A workaround is to store large tables of pre-computed digits in a secure location. The number of quadratic irrationals you can choose from is infinite. Also, your digit sequence should never start with the first binary digit of such numbers, but rather at a random position, to make hacking more difficult.

For instance, to generate your sequence $\{d(k)\}$, set $d(3k)$ to $\delta(g_1(k), \alpha_1)$, set $d(3k+1)$ to $\delta(g_2(k), \alpha_2)$, and set $d(3k+2)$ to $\delta(g_3(k), \alpha_3)$ where

- The numbers $\alpha_1, \alpha_1, \alpha_3$ are three quadratic irrationals, say $\sqrt{2}, \sqrt{10}, \sqrt{41}$,
- The number $(\delta(k, \alpha) + 1)/2$ is the k-th binary digit of the quadratic irrational α,
- The functions g_1, g_2, g_3 are used for scrambling: for instance, $g_1(k) = 5 \cdot 10^5 + 2k$, $g_2(k) = 3 \cdot 10^5 + 3k$, and $g_3(k) = 7 \cdot 10^6 - k$.

Another solution is to use for your sequence $\{d(k)\}$ a bitwise XOR [Wiki] on two pseudo-random sequences: the binary digits of (say) $\sqrt{2}$ and $\sqrt{41}$, starting at arbitrary positions.

There are also Python libraries that provide solutions suitable for cryptographic applications. For instance, `os.urandom()` uses the operating system to create random sequences that can not be seeded, and are thus not replicable. See here and here.

3.2 Prime test implementation to detect subtle flaws in PRNG's

The code presented here performs the prime test, computing $L_3(n)$. The variable `nterms` represents n, and it is set to 10,000. Rather than directly computing $L_3(n)$, the code iteratively computes more granular statistics, namely `minL` and `maxL`; $L_3(n)$ is the maximum between `-minL` and `maxL`, obtained at the last iteration.

The seeds and the sequences are initialized in the main part, at the bottom. The default category `nonSquare` is used for L_3. The other categories, `Prime` and `All`, are respectively for L_1 defined in For-

mula (5) and L_4 defined in Formula (6). If you use the function `createSignHash()` rather than the default `createSignHash2()`, you can easily compute L_2. The code is somewhat long only because it covers all the options discussed in section 2.2, and more. It heavily relies on hash tables (dictionaries in Python) rather than arrays, because the corresponding arrays would be rather sparse, consume a lot of memory, and slow down the computations. In addition, the code can easily handle Beurling primes (non-integer primes) thanks to the hash tables. A lengthier version named `dirichletL.py`, computing the orbit of $L_P(z, \chi)$ for z in the complex plane when $\sigma = \Re(z) \geq \frac{1}{2}$ is fixed, for any set of primes P (finite or infinite) including Beurling primes, is available on GitHub, here.

The Python code does not use any exotic library other than `primePy`. To install this library, type in the command `pip install primePy` on the Windows Command Prompt or its Unix equivalent, as you would to install any library. There is a possibility that some older versions of Python would require the `BigNumber` library. The code was tested under Python 3.10. The source code, featured below, is also on GitHub: look for `randomNumbersTesting.py`.

```
# Test randomness of binary sequences via the law of the iterated logarithm
# By Vincent Granville, www.MLTechniques.com

import math
import random
import numpy as np
from primePy import primes

#--
def createRandomDigits(method,seed):
  primeSign={}
  idx=0
  if method=='SQRT':
    y=seed[0]
    z=seed[1]
  elif method=='Python':
    random.seed(seed)
  else:
    x=seed
  start=2
  if method=='Dirichlet4':
    start=3
  for k in range(start,nterms):
    if k%2500==0:
      print(k,"/",nterms)
    if primes.check(k):
      primeSign[k]=1
      if method=='SQRT':
        if z<2*y:
          y=4*y-2*z
          z=2*z+3
        else:
          y=4*y
          z=2*z-1
          primeSign[k]=-1
      elif method=='Dirichlet4':
        if k%4==seed:
          primeSign[k]=-1
      elif method=='CounterExample':
        idx=idx+1
        if idx%2==seed:
          primeSign[k]=-1
      elif method=='Python':
        x=random.random()
      elif method=='Logistic':
        x=4*x*(1-x)
      elif method=='Base3':
        x=3*x-int(3*x)
      if method in ('Python','Logistic','Base3') and x>0.5:
        primeSign[k]=-1
```

```
  return(primeSign)

#--
def createSignHash2():
  signHash={}
  signHash[1]=1
  for p in primeSign:
    oldSignHash={}
    for k in signHash:
      oldSignHash[k]=signHash[k]
    for k in oldSignHash:
      pp=1
      power=0
      localProduct=oldSignHash[k]
      while k*p*pp<nterms:
        pp=p*pp
        power=power+1
        new_k=k*pp
        localProduct=localProduct*primeSign[p]
        signHash[new_k]=localProduct
  return(signHash)

#--
def createSignHash():
  # same as createSignHash() but for square-free integers only
  signHash={}
  signHash[1]=1
  for p in primeSign:
    oldSignHash={}
    for k in signHash:
      oldSignHash[k]=signHash[k]
    for k in oldSignHash:
      if k*p<nterms:
        new_k=k*p
        signHash[new_k]=oldSignHash[k]*primeSign[p]
  return(signHash)

#--
def testRandomness(category):
  signHash=createSignHash2()
  isSquare={}
  sqr=int(math.sqrt(nterms))
  for k in range(sqr):
    isSquare[k*k]=1
  count=0
  count1=0
  sumL=0
  minL= 2*nterms
  maxL=-2*nterms
  argMin=-1
  argMax=-1
  for k in sorted(signHash):
    selected=False
    if category=='Prime' and k in primeSign:
      selected=True
    elif category=='nonSquare' and k not in isSquare:
      selected=True
    elif category=='All':
      selected=True
    if selected==True:
      if signHash[k]==1:
        count1=count1+1
      count=count+1
      sumL=sumL+signHash[k]
      if sumL<minL:
        minL=sumL
```

```
      argMin=count
    if sumL>maxL:
      maxL=sumL
      argMax=count
  return(minL,argMin,maxL,argMax,count,count1)

#--
# Main Part. Requirements:
#  0 < seed < 1 for 'Base3' and 'Logistic'; rational numbers not random
#  seed=(y,z) with z>y, z!=2y, y!=2x and x,y>0 are integers for 'SQRT'
#  swapping -1/+1 for seed=(90,91) in 'SQRT' does well, the original does not

seedMethod={}
seedMethod['Python']=(0,1,2,4,100,200,500)
seedMethod['Logistic']=(0.181517,0.72)
seedMethod['Base3']=(0.181517,0.72)
seedMethod['SQRT']=((2,5),(90,91))
seedMethod['Dirichlet4']=(1,3)
seedMethod['CounterExample']=(1,0)
categoryList=('Prime','nonSquare','All')

nterms=10000

OUT=open("prngTest.txt", "w")
for method in seedMethod:
  for seed in seedMethod[method]:
    for category in categoryList:

     primeSign=createRandomDigits(method,seed)
     [minL,argMin,maxL,argMax,count,count1]=testRandomness(category)

     string1=("%14s %9s|%5d %5d|%5d %5d|%5d %5d|" % (method,category,\
       minL,maxL,argMin,argMax,count1,count))+str(seed)
     print(string1)
     string2=("%s\t%s\t%d\t%d\t%d\t%d\t%d\t%d\t" % (method,category,\
       minL,maxL,argMin,argMax,count1,count))+str(seed)+'\n'
     OUT.write(string2)

OUT.close()
```

3.3 Special formula to compute 10 million digits of $\sqrt{2}$

The purpose of this code is twofold: to show you how to process integers with millions of digits in Python, and to offer a simple mechanism to compute the binary digits of some quadratic irrational numbers such as $\sqrt{2}/2$. The first problem is solved transparently with no special code or library in Python 3.10. In short, this is a non-issue. With older versions of Python, you might have to install the `BigNumber` library. See documentation here. Nevertheless, it would be a good idea to track the size of the integers that you are working with (`y` and `z` in my code), as eventually their size will become the bottleneck, slowing down the computations.

As for the actual computation of the digits, it is limited here to 10,000 digits, but I compare these digits with those obtained from an external source: Sagemath, see here. It shows, as it should, that both methods produce the same digits, for the number $\sqrt{2}/2$ in particular.

The special recursion used for the digit computation is as follows:

If $z_k < 2y_k$ **then**
$y_{k+1} = 4y_k - 2z_k$
$z_{k+1} = 2z_k + 3$
$d(k) = 1$
else
$y_{k+1} = 4y_k$
$z_{k+1} = 2z_k - 1$
$d(k) = 0.$

The bivariate seed (the initial condition) is determined by the values of y_0 and z_0. You need $z_0 > y_0$ and

$z_0 \neq 2y_0$. Then the binary digits $d(k)$ are those of the number

$$x_0 = \frac{-(z_0-1) + \sqrt{(z_0-1)^2 + 8y_0}}{4},$$

see here. In particular, if $y_0 = 2, z_0 = 5$, then $x_0 = -1 + \sqrt{2}$. Using the change of variables $u_k = 2y_k - z_k$ and $v_k = 2z_k + 3$, the recurrence can be rewritten as:

If $u_k > 0$ **then**
  $u_{k+1} = 4u_k - v_k$
  $v_{k+1} = 2v_k + 3$
  $d(k) = 1$
else
  $u_{k+1} = 4u_k + v_k - 2$
  $v_{k+1} = 2v_k - 5$
  $d(k) = 0$.

Now $v_k - 5$ is divisible by 8. Let $w_k = (v_k - 5)/8$. We have $d(k) = 1$ if w_{k+1} is odd, otherwise $d(k) = 0$. We also have the following one-dimensional backward recursion, allowing you to compute the digits backward all the way down to the first digit:

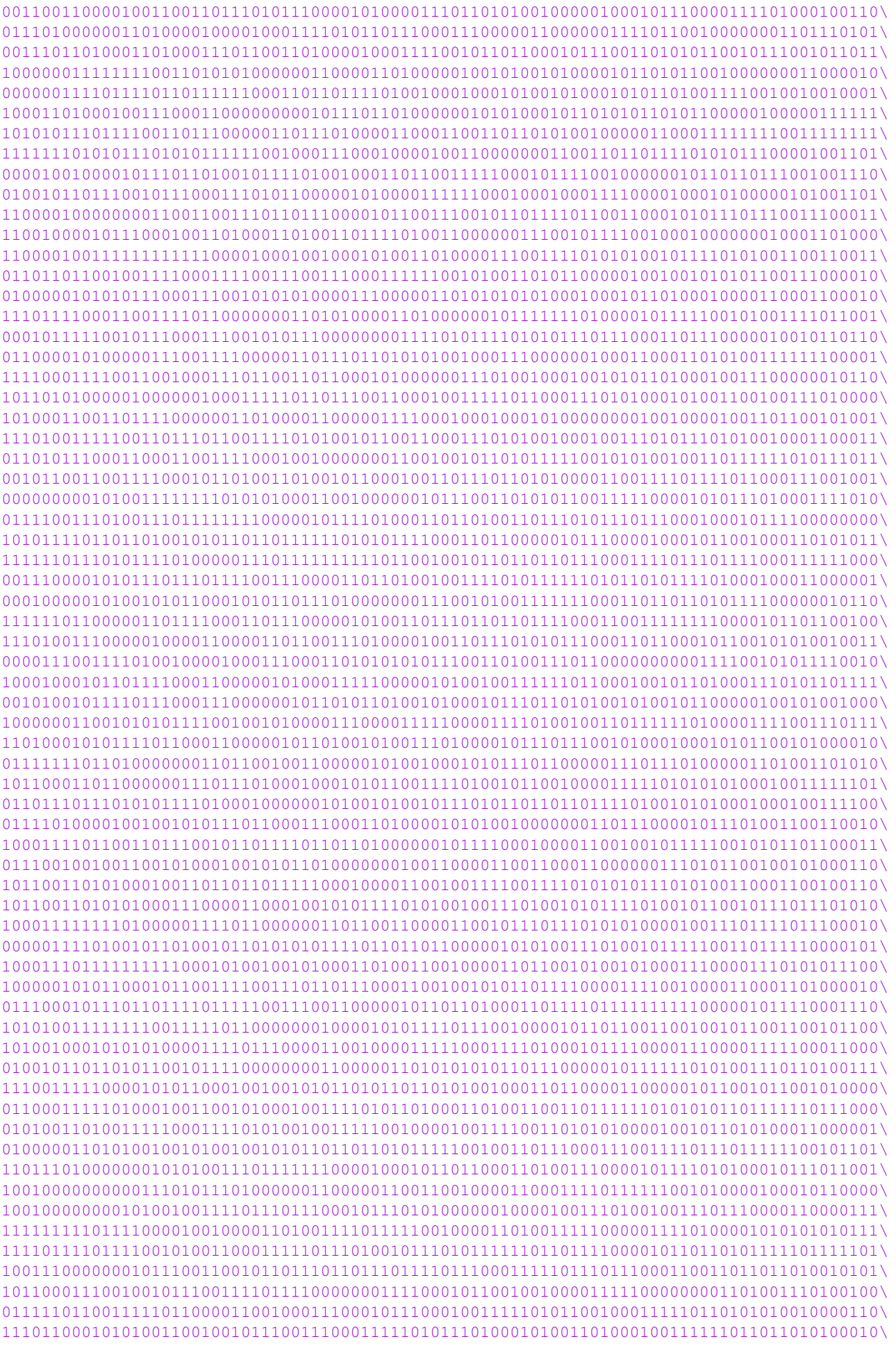
If w_{k+1} is odd, **then**
  $v_k = (v_{k+1} - 3)/2$
  $d(k) = 1$
else
  $v_k = (v_{k+1} + 5)/2$
  $d(k) = 0$.

These recursions are reminiscent of the unsolved Collatz conjecture [Wiki]. Below is the source code, also available on GitHub: look for `randomNumbers-sqrt2.py`.

```
# Comparing binary digits of SQRT(2) obtained with two different methods

# Method 1:
# 10,000 binary digits of SQRT(2) obtained via https://mltblog.com/3uMZQ4s
# Using sagemath.org. Sagemath commmand: N(sqrt(2),prec=10000).str(base=2)

sqrt2='01101010000010011110011001100111111100111011110011001001000010001011001011111011\
00100111011001101110101010010101011111010011111000111010111011110110000010111010100010010\
11101110101000010011001110110100010111101011001000010110000011001100111001100100010101\
00101011111100100000110000010000111010101110001010001011000011101010001011000111111111001\
10111111011100100000111101101100111001000011110111010010101000010111100100001110011100001\
11101101001010011110000000001001000011100110110001111011111110100010011101101000110100100\
10000000101110100001110100001010101111000111110100111001010011000001011001110001100000000\
01000110111100001100110111101111001010101100011011110010010001000101101000100001000100110\
00101001000110000010101011110001110010001011110111111000100111000110011110001101101010110\
10100010100011100010111011011111101001110111001100101100101010011000110100001100110001111\
11001111001000010011011111010100101111000100100000111110000011011011100101100000101110111\
10101010100100101000001000100110010000010000001100101001001010100000010011100101001010100\
11011011011000111111010000111011111101111101001101001110100000001011001110101111001001001\
11111000001100010000100110010011011010101111001101010100101000101101100101000110111000011\
00111100110100000110110110111110000010001101101100011100000001000001001101110000000000111\
11111000110010001101010010111100110011001010100101111010011111011110111011110110100001111010\
11111111110110101000011011111000111111111001010001000100001001100000111110111101010000001\
10001001000001111101111010101010000001110000101100000011111110010111101110111101010001011\
11011111011100001100110001100010000011100010100010111010101111111101011111001110110010011\
01001001001111010010100111011000111111101011001011100010000010111111011111111000010111100\
00111111010011101100011111011110000000111110011010110011001110010000010111100101111111100\
10100000000101110000100100011001111000010111101001001010010101011100000010001101011111111\
00111110001111011110111100101000111110101000110011101101001101011111000110000010100101110\
10011000110011111000111110000101001000010111110011101101001001010011101101100000101011110001\
10000010000101101011000010011111011010010000111110010010010011111010110110111000111111\
00000101101110011010010100100000011011000001001110111101110100010010001001010011000001111\
10101001110101010101101000011101110101000110010000111110111010010001001111101000110101000\
11111101001000000110000101111100010000011111011011101001010011101111101101011001100011001\
```



```
1101111110001010110111110100001111010101110011010101001010110000111010011111011010100000\
0011111011010011011000101111011011001101011100000011101111110111010000100011110110101010\
0100100000101011101011111101110001111100101001111111001010110000101100101000110101110110\
0100000100000010110000101111000000001011001100101101000000001011011101101001111100111011\
0100111100101101111011000110011110010011101000100111111011011000001011100101100001110101\
1111110001101000000001101000110010111010011001110111010110101000100100011111100010011111\
0010110000000001010001111110101110000001001100110010001101001100101010101001010001101010\
0000000000011111111110000010010000101010111111011110000101101100010100100101010001000\
0\
0010100010110011110110100010010011110100101010011001100111011001011011000111100111101100\
0110111001110100111000100011100100111001111100100010101011001101100001110010111101100101\
0000010011110001010011111110000101110111000101110111110001000101000101101101110110010101\
0110000111101101110100101010101011101000100100011111001001111101110101110000000111101101\
0011111001101001011010100100101110001110110000011110011001110000000011101111100111011011\
0111000101011001110111011111010100011011011001111000111110011100010011011100100001010101\
0100101100110000100110101001101101110010101011000010010111011000101001010001010001001000\
1011111110011110100001111011011001101110101011111110000101000001010000111111100001001001\
1111110001000011000111101001011011000110000011100100000100100000100000001000100101100100\
0110010111100111000110001000000001100101111000111000011101010111000100101110001110111010\
1001111011110010110011011011101100101110110000101001011101011101000110101111001011110100\
0001000010010001011000111010101110011111101101100010011011111010001'

size=len(sqrt2)

# Method2:
# 10,000 binary digits of SQRT(2) obtained via formula at https://mltblog.com/3REtOB9
# Implicitly uses the BigNumber Python library (https://pypi.org/project/BigNumber/)

y=2
z=5
for k in range(0,size-1):
  if z<2*y:
    y=4*y-2*z
    z=2*z+3
    digit=1
  else:
    y=4*y
    z=2*z-1
    digit=0
  print(k,digit,sqrt2[k])
```

References

[1] Keith Conrad. *L-functions and the Riemann Hypothesis*. 2018. 2018 CTNT Summer School [Link]. 4

[2] Harold G. Diamond and Wen-Bin Zhang. *Beurling Generalized Numbers*. American Mathematical Society, 2016. Mathematical Surveys and Monographs, Volume 213 [Link]. 5

[3] Vincent Granville. *Applied Stochastic Processes, Chaos Modeling, and Probabilistic Properties of Numeration Systems*. MLTechniques.com, 2018. [Link]. 5

[4] Adam J. Harper. Moments of random multiplicative functions, II: High moments. *Algebra and Number Theory*, 13(10):2277–2321, 2019. [Link]. 1

[5] Adam J. Harper. Moments of random multiplicative functions, I: Low moments, better than squareroot cancellation, and critical multiplicative chaos. *Forum of Mathematics, Pi*, 8:1–95, 2020. [Link]. 1, 3

[6] Adam J. Harper. Almost sure large fluctuations of random multiplicative functions. *Preprint*, pages 1–38, 2021. arXiv [Link]. 4

[7] T. W. Hilberdink and M. L. Lapidus. Beurling Zeta functions, generalised primes, and fractal membranes. *Preprint*, pages 1–31, 2004. arXiv [Link]. 4, 5

[8] Yuk-Kam Lau, Gerald Tenenbaum, and Jie Wu. On mean values of random multiplicative functions. *Proceedings of the American Mathematical Society*, 142(2):409–420, 2013. [Link]. 1, 3

[9] E.C. Titchmarsh and D.R. Heath-Brown. *The Theory of the Riemann Zeta-Function*. Oxford Science Publications, second edition, 1987. 4